FERMILAB-Conf-90/87

Cooperative Processes Software (CPS)*

Chip Kaliher
Fermi National Accelerator Laboratory
P.O. Box 500
Batavia, Illinois 60510

April 1990

* Presented at the 1990 Conference on Computing in High Energy Physics, Santa Fe, New Mexico, April 9-13, 1990.

COOPERATIVE PROCESSES SOFTWARE (CPS)

Chip Kaliher
Fermi National Accelerator Laboratory
Batavia, IL

ABSTRACT

CPS is a package of software tools for splitting a computational task, called a job, among a set of processes distributed over one or more processors. It is designed to function as a tool for solving computing problems which require many computing cycles per I/O byte, and is well suited for computing platforms consisting of "farms" of processors, operating in parallel. This paper describes three essential features of CPS: data transfers between cooperating processes, remote subroutine calls, and process queues.

INTRODUCTION

CPS is a software product developed by the Computing R&D Department (formerly known as the ACP Dept) at Fermilab. It is a package of software tools that make it easy to split a computational task, called a job, among a set of processes, operating on one or more processors. Apart from considerations of execution speed, the processes operate identically executing on one single processor, or on multiple processors. Each process executes its own private copy of the user's program. See Figure 1.

CPS is designed to function as a software tool for solving a certain class of computing problems, especially those which require many computing instructions and operations per I/O byte. The software is most useful for problems in which there is a "course granularity" (event parallelism) of the input data, and is well suited for computing platforms consisting of farms of processors, operating in parallel.

Much HEP event reconstruction computing is done using offline RISC-based processor farms, running various flavors of the UNIX operating system. Typically, these problems consist of many independent, uncorrelated physics events, (>200 GB), and often require more raw computing capacity than is available in a single "box." The necessity of using multiple processors of course implies the use of multiple processes, (at least one process per processor). Multiple processes can be configured independently or can be either loosley or tightly coupled (Figures 2, 3, 4). Topology C often has several advantages over Topology A and/or Topology B. Fewer external media are required. It is considerably easier to manage. It is also faster and more efficient, since there are fewer data moves, and the number of processes in each class can be optimized for each problem. Finally, it is integrated. All the processors function as a single, logical compute server, working together on a problem.

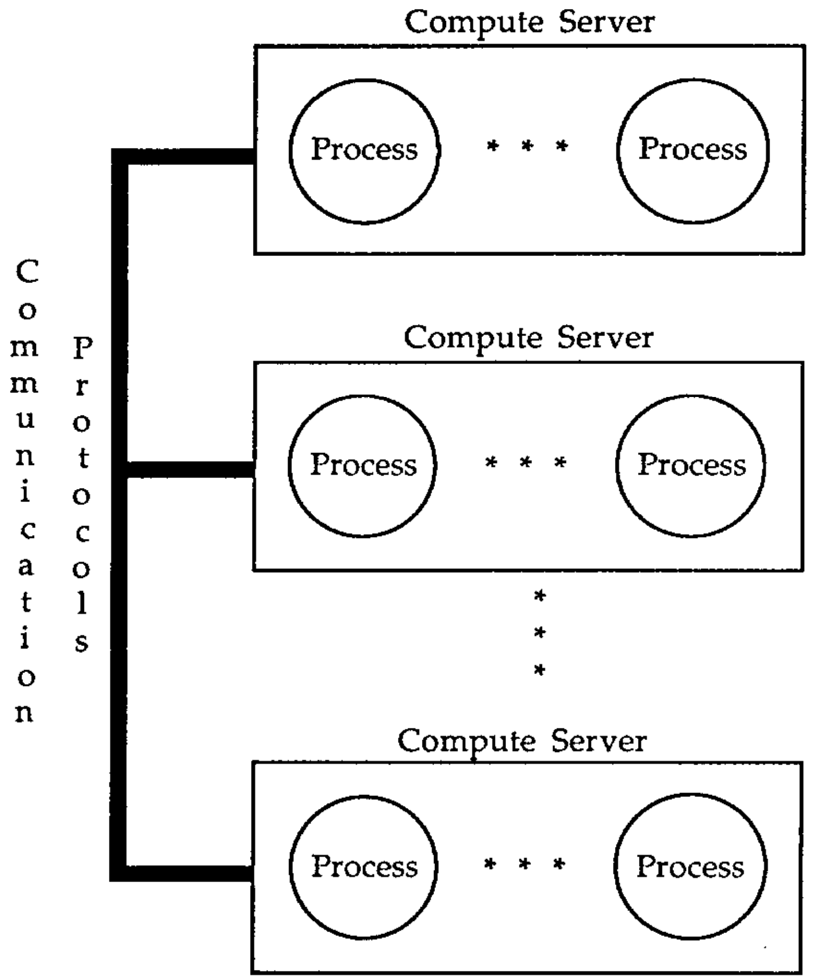

Figure 1. CPS Process Distribution

Topology C is often referred to as the cannonical event reconstruction example. There are three classes or ranks of processes. All processes within a class run the same program. The class 1 process gets an input event from some external media (e.g. 8mm tape), then dequeues a class 2 process from the ready queue. Next, the class 1 process transfers the event data from its own virtual address space, to the address space of the class 2 process. Finally, the class 1 process issues a remote subroutine call to the class 2 process, causing the latter to perform the computing operations necessary to reconstruct the event. As part of the call, the class 2 process is directed to place itself on the done queue when the remote subroutine call completes.

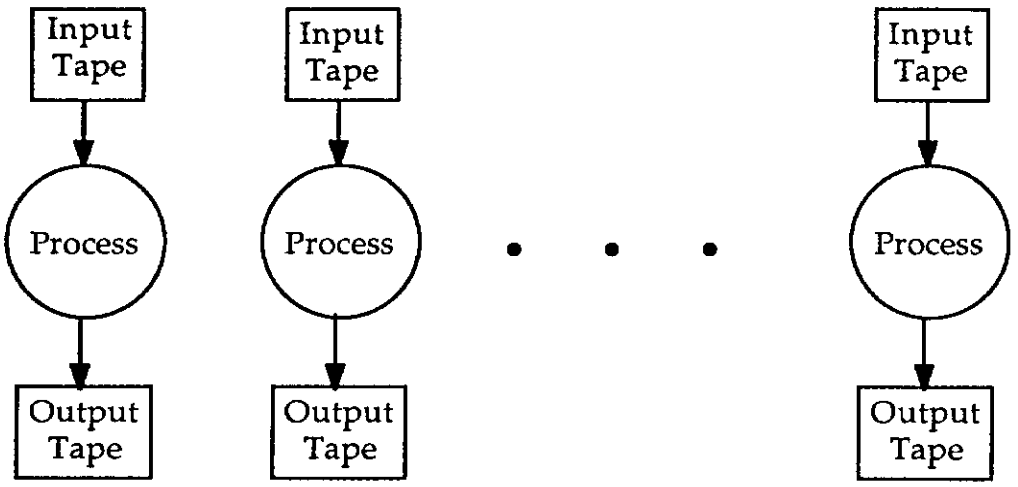

Figure 2. Topology A

CPS EXAMPLE

The class 1 process repeats this sequence of operations: get an input event, dequeue a class 2 process, transfer the data, call a remote subroutine, until the supply of input events has been exhausted, at which point it will initiate the end-of-job operations (summation, histogramming, etc). If it is unable to dequeue a class 2 process at any point during the job, (the ready queue is empty), the class 1 process simply waits until a class 2 process become available. The class 2 processes receive blocks of event data, reconstruct the event, then enter the done queue, as directed by the class 1 process. The class 3 process dequeues a class two process from the done queue, then transfers the reconstructed event data from the latter's virtual address space to its own. It re-queues the class 2 process back to the ready queue, then puts the reconstructed event data onto some external output medium (8mm tape). The class 3 process repeats this sequence of operations: dequeue a class 2 process, transfer the data, re-queue the class 2 process, write the output reconstructed event, until it encounters an end-of-queue condition, signalling the end-of-job. At the start of each job, after performing all the required declaration and initialization functions, all the processes in the job synchronize with each other, to establish reliable cooperation.

CPS TOOLS

Each CPS process calls routines for initialization and declaration. A process first calls the acp_init routine, to establish the necessary communication links, initialize local variables, etc. If a process will serve as a source or destination of data for block transfers between cooperating processes, the process must call acp_declare_block, specifying the address, length in bytes, and the block number for each transferrable data block in the process virtual address space. If the process will be placed on process queues, it must call the acp_declare_queue routine, specifying the queue number, and optional queue arguments, for each process queue on which the process may be placed. If the process will function as a server, allowing other processes to issue remote subroutine calls to its local routines, it must call the routine, acp_declare_subroutine, specifying the entry point address, the remote subroutine number, the number of arguments, and their individual byte counts, for each of the process local routines which will be remotely callable. Lastly, each CPS process calls acp_sync, to allow all processes in all job classes to complete their various individual process declarations, before real cooperative processing begins.

A process which will function as the destination of a block transfer could declare the destination of the transfer as follows:

```
integer*4 destination(1000)
. . .
call acp_declare_block(destination,4000,23)
```

where 4000 is the length in bytes of the destination block, and 23 is a unique number that another process can use to refer to that block. Another process could send a block of data as follows:

```
integer*4 source(1000)
. . .
call acp_send(process,source,4000,23,0)
```

where process is the number of the process to which the data is to be sent, 4000 is the number of bytes to be sent, 23 is the number of the block where the data will be sent, and 0 is the offset within the destination block where the first byte of the transfered data is placed. A server process could place itself on a process queue as follows:

```
call acp_declare_queue(ACP$THIS_PROCESS,27)
```

where 27 is the queue number. A client process could dequeue the server process as follows:

```
call acp_dequeue_process(process,27)
```

where process is a return argument which will contain the process number of the dequeued process when the call completes and 27 is the queue number from which the process is to be dequeued. A server process could declare a remote subroutine as follows:

```
program server
external x
. . .
call acp_declare_subroutine(x,73,2, 4,4)
. . .
call acp_service_calls
end
. . .
subroutine x
integer*4 a, b
. . .
return
end
```

where x is the remote subroutine which is declared, 73 is a unique number that another process can use to refer to that subroutine, and there are two arguments, each 4 bytes long. A client process could call the remote subroutine as follows:

```
integer*4 a, b
. . .
call acp_call(process,action,73,a,b)
```

where process is the process number of the server process, action is wait, no-wait, or a queue number, 73 is the number of the remote subroutine which is called, and a and b are the arguments.

JOB MANAGER

The Job Manager reads the user's job description file (JDF) at the start of the job. This file specifies the number of processes to be created in each job class, the name/location of the program which the processes in each class will run, and the type of processor on which the processes in each class are to be created. The Job Manager starts all the required processes, creates and manages the process queues required by the job, and provides I/O services (tape mounts, etc). The Job Manager monitors each process in the job (Figure 5), and stops all the processes at the end of the job.

HARDWARE

CPS is supported on MIPS M500 and M120 systems, and also on Silicon Graphics SGI 4D/xxx systems. It has been test ported to DEC VAX/VMS, DEC ULTRIX, SUN and Apollo systems.

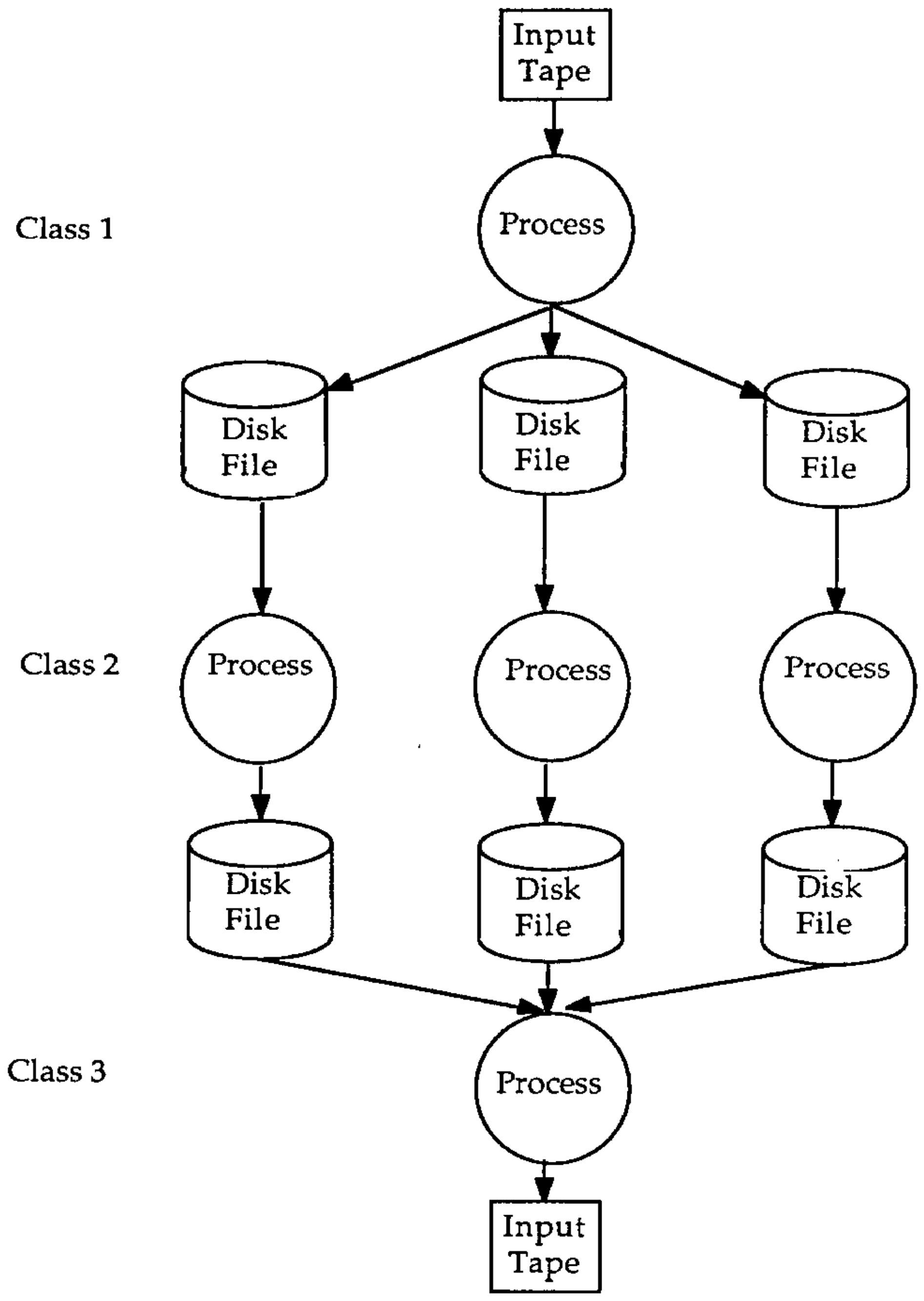

Figure 3. Topology B

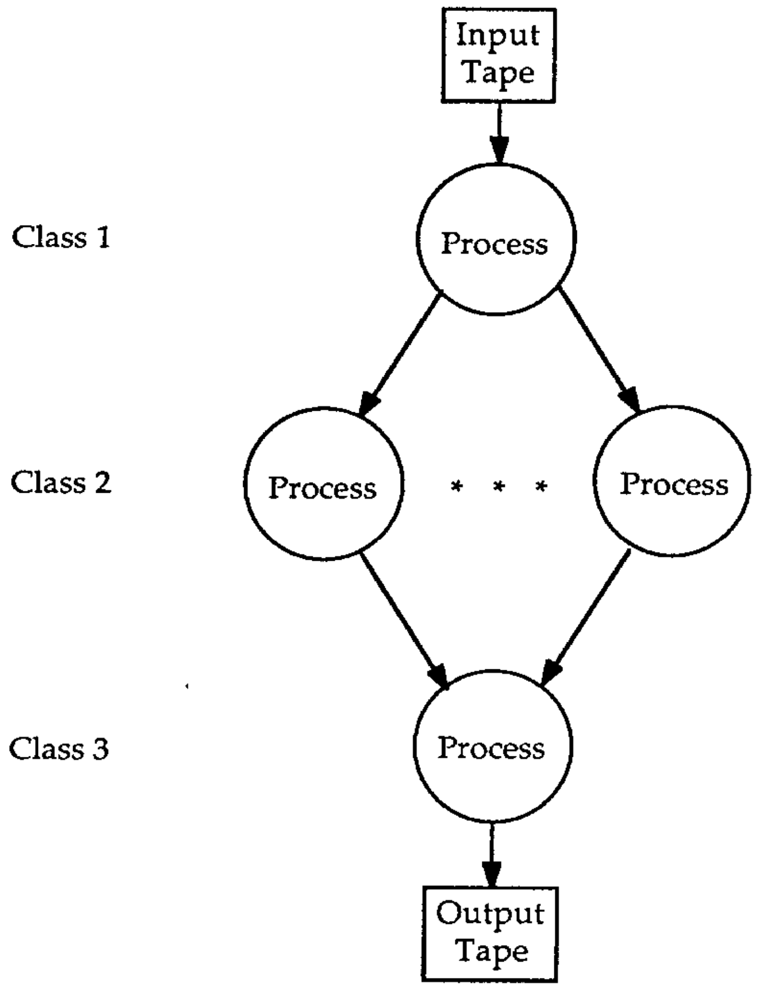

Figure 4. Topology C

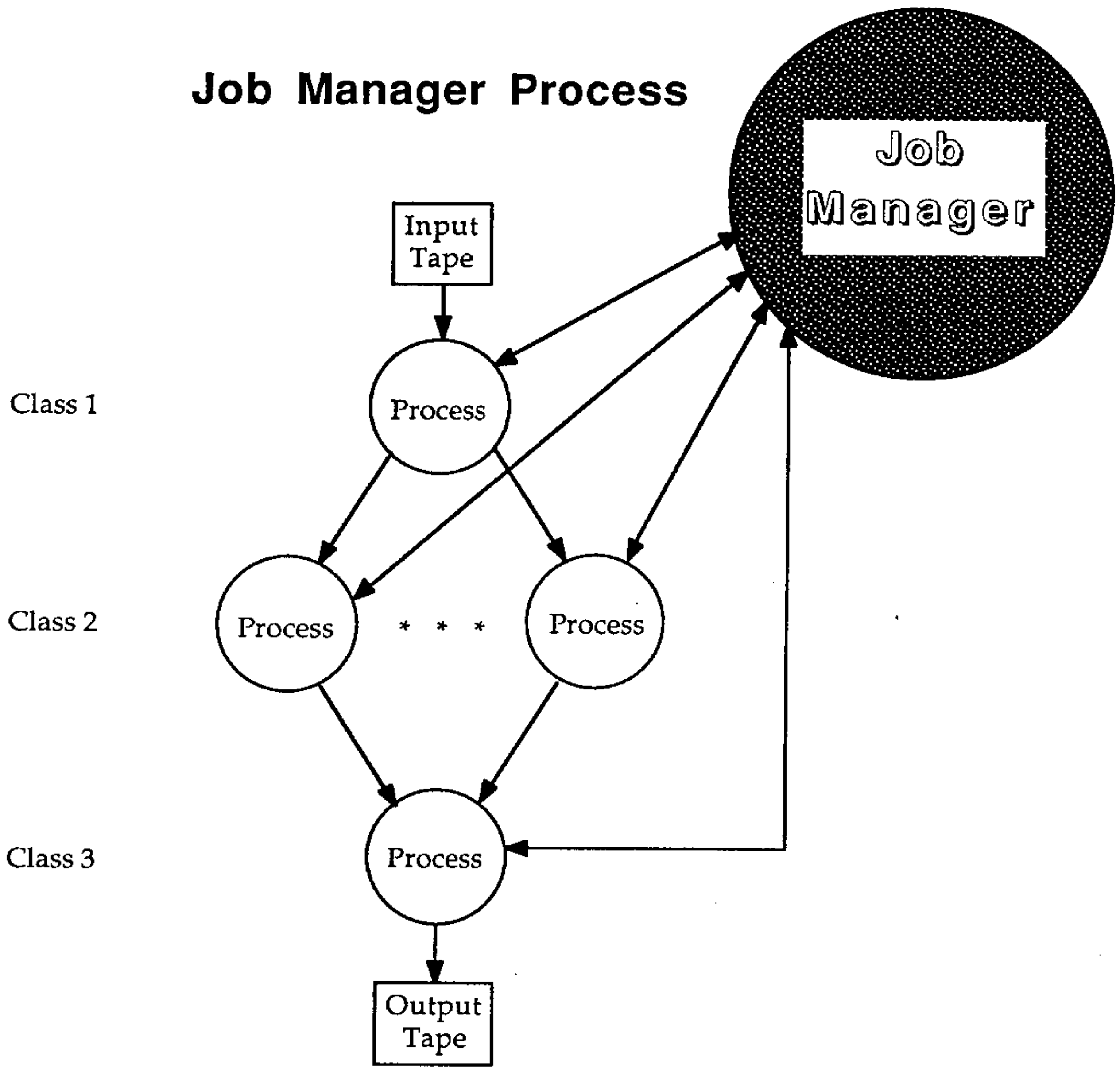

Figure 5. Job Manager